An Empirical Study of Delta Algorithms

James J. Hunt[1], Kiem-Phong Vo[2] and Walter F. Tichy[1]

[1] University of Karlsruhe, Karlsruhe, Germany
[2] AT&T Research, Murray Hill, NJ

Abstract. Delta algorithms compress data by encoding one file in terms of another. This type of compression is useful in a number of situations: storing multiple versions of data, distributing updates, storing backups, transmitting video sequences, and others. This paper studies the performance parameters of several delta algorithms, using a benchmark of over 1300 pairs of files taken from two successive releases of GNU software. Results indicate that modern delta compression algorithms based on Ziv-Lempel techniques significantly outperform *diff*, a popular but older delta compressor, in terms of compression ratio. The modern compressors also correlate better with the actual difference between files; one of them is even faster than *diff* in both compression and decompression speed.

1 Introduction

Delta algorithms, i.e., algorithms that compute differences between two files or strings, have a number of uses when multiple versions of data objects must be stored, transmitted, or processed. Differencing compression is an essential ingredient of most software configuration management systems. Without efficient differencing algorithms, version tracking would be impractical for most applications.

Most uses stem from the fact that a delta is often one to two orders of magnitude smaller than the original, and significantly smaller than a direct compression of the original. For example, version control systems store multiple versions of programs, graphics, documents, and other data as deltas relative to a base version[10, 12]. Similarly, backup programs can save space by storing deltas. Checkpoints of large data spaces can be compressed dramatically by using deltas and can then be reloaded rapidly. Display updates can also be performed efficiently using a delta that exploits operations that move lines around[1]. Furthermore, changes of programs or data are most economically distributed as update scripts that generate the new data from the old. Transmitting the script, which is nothing but a delta, can save space, time, and network bandwidth. In addition, a delta provides an effective form of encoding; only the holder of the original can successfully generate the new version. This facility is becoming especially interesting in the Internet for distributing software updates.

Other uses of deltas are for highlighting the differences between two versions of a program or document and for merging or reconciling competing changes of a common original. Deltas are also needed for sequence comparison in molecular

biology. The main focus of this paper, however, is the size of the deltas that various delta algorithms compute and the speed of compression and decompression.

The state of empirical comparisons of delta algorithms is poor. Miller and Myers[7] compare the runtime of their delta program, *fcomp*, with that of UNIX *diff* [2, 3]. Their first test involves two pairs of (highly untypical) files, and *fcomp* fails on one of them. Additional tests were run, but not enough particulars are given to repeat the tests independently. Obst[9] compares several difference algorithms on programs of about 3 Megabytes. No details are given that would permit the repetition of their experiment. In both instances, the claims are quite doubtful. The unreliability of the observations is underscored by outliers and irregularities.

The purpose of this paper is to both suggest a realistic benchmark for comparing the performance of delta algorithms and to present firm performance results for a set of such algorithms. The intent is to make the study reported here repeatable by anyone who wishes to check or extend the results. We also present a new delta algorithm called *vdelta* developed by David Korn and Phong Vo [13, 5].

2 Benchmark

The authors propose using the Longest Common Subsequence (LCS) as the reference against which to measure the effectiveness of a differencing algorithm applied to one dimensional data. In particular, they define the **difference** between two file as the average size of the two files minus the LCS. This yardstick is used to build a benchmark consisting of all files in two successive releases of Gnu *emacs*, releases 19.28 and 19.29, and of GNU *gcc*, releases 2.7.0 and 2.7.1. This benchmark contains 810 text files (C programs, Lisp programs, documentation) and 300 files with lisp byte code (binary files). The authors also compiled the 201 C program files present in both versions and included them in the study. The authors chose this benchmark because it is freely available and offers a large number of files of various types in successive revisions.

3 Algorithms

There are four differencing algorithms that are used for this study. The first—*longest common subsequence*—is used as a yardstick to measure the effectiveness of the other three because it is an exhaustive algorithm for finding the Longest Common Subsequences of any pair of files. UNIX *diff* finds an approximation of the Longest Common Subsequence by considering whole lines instead of characters as indivisible units. The last two—*bdiff*, and *vdelta* — piece together the second file out of blocks from the first file. Unlike *lcs*, these algorithms take the ordering of blocks into account. All three of these algorithms run enough faster than *lcs* to have practical applications. Both *bdiff* and *vdelta* offer additional compression on the resultant delta. For this reason, *diff* is also compared with *gzip* post processing.

3.1 *Longest Common Subsequence*

The *longest common subsequence* algorithm (*lcs*) is a textbook algorithm applied to strings[3, 8]. Its runtime is $O(nm)$ where n and m are the files sizes, so it is not practical for general use. However, it is a good point of reference, since it is guaranteed to find the Longest Common Subsequence of two linear character sequences. The **difference** between two files can be expressed as the mean size of the two files minus the size of the LCS.

3.2 UNIX *diff*

UNIX *diff* uses the *Longest Common Subsequence* algorithm computed on a line-by-line basis instead of a character-by-character basis[2]. It is much faster than *lcs* because it does not examine all possible combinations of characters. Only common lines can be found with *diff*. Since *diff* only produces output for text files, the contents of binary files must be folded into the ASCII printable range. A commonly used tool for this is *uuencode*.

3.3 *Bdiff*

Bdiff is a modification of W. F. Tichy's block-move algorithm[11]. It uses a two-stage approach. First it computes the difference between the two files. Then it uses a second step to compress the resulting difference description. These two parts run concurrently in that the first stage calls the second each time it generates output.

In the first phase, *bdiff* builds an index, called a suffix tree, for the first file. This tree is used to look up blocks, i.e. substrings, of the second file to find matches in the first file. A greedy strategy is used, i.e. every possible match is examined, to ensure that the longest possible match is found. The output from this phase is a sequence of copy blocks and character insertions that encode the second file in terms of the first. It can be shown that the algorithm produces the smallest number of blocks and runs in linear time. It also discovers crossing blocks, i.e., blocks whose order was permuted in the second file.

The second phase efficiently encodes the output of the first. A block is represented as a length and an offset into the first file. Characters and block lengths are encoded in the same space by adding 253 (256 minus the three unused lengths) to lengths before encoding. Blocks of lengths less than 4 are converted to character insertions. Characters and lengths are then encoded using a common splay tree[4]. The splay tree is used to generate a character encoding that ensures that frequently encoded characters are shorter than uncommon characters. Splay trees dynamically adapt to the statistics of the source without requiring an extra pass. A separate splay tree encodes the offsets.

Bdiff actually uses a sliding window of 64 KB on the first file, moving it in 16 KB increments. This means that the first phase actually builds four suffix trees that index 16 KB each of the first file. The window is shifted forward whenever the encoding of the second file crosses a 16 KB boundary, but in such a fashion

that the top window position in the first file is always at least 16 KB ahead of the current encoding position in the second file. Whenever the window is shifted, the oldest of the four suffix trees is discarded and a new one built in its space. The decoder has to track the window shifts, but does not need to build the suffix trees. Position information is given as an offset from the beginning of the window.

3.4 *Vdelta*

Vdelta is a new technique that combines both data compression and data differencing. It is a refinement of W. F. Tichy's block-move algorithm[11], in that, instead of a suffix tree, *vdelta* uses a hash table approach inspired by the data parsing scheme in the 1978 Ziv-Lempel compression technique [14]. Like block-move, the Ziv-Lempel technique is also based on a greedy approach in which the input string is parsed by longest matches to previously seen data. Both Ziv-Lempel and block-move techniques have linear-time implementations [6]. However, implementations of both of these algorithms can be memory intensive and, without careful consideration, they can also be slow because the work required at each iteration is large. *Vdelta* generalizes Ziv-Lempel and block-move by allowing for string matching to be done both within the target data and between a source data and a target data. For efficiency, *vdelta* relaxes the greedy parsing rule so that matching prefixes are not always maximally long. This allows the construction of a simple string matching technique that runs efficiently and requires minimal main memory.

Building Difference For encoding, data differencing can be thought of as compression, where the compression algorithm is run over both sequences but output is only generated for the second sequence. The idea is to construct a hash table with enough indexes into the sequence for fast string matching. Each index is a position which is keyed by the four bytes starting at that position. In order to break a sequence into fragments and construct the necessary hash table, the sequence is processed from start to end; at each step the hash table is searched to find a match. Processing continues at each step as follows:

1. if there is no match,
   (a) insert and index for the current position into the hash table,
   (b) move the current position forward by 1, and
   (c) generate an insert when in output mode; or
2. if there is a match,
   (a) insert into the hash table indexes for the last 3 positions of the matched portion,
   (b) move the current position forward by the length of the match, and
   (c) generate a copy block when in output mode.

Each comparison is done by looking at the last three bytes of the current match plus one unmatched byte and checking to see if there is an index in the hash table

that corresponds to a match. The new match candidate is checked backward to make sure that it is a real match before matching forward to extend the matched sequence. If there is no current match, i.e. just starting a new match, use the 4 bytes starting at the current position.

As an example, assume the sequence below with the beginning state as indicated (the ⇓ indicates the current position):

```
⇓
0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19
b  c  d  e  a  b  c  d  a  b  c  d  a  b  c  d  e  f  g  h
```

The algorithm starts at position 0. At this point the rest of the sequence is the entire sequence so there is no possible match to the left. Case 1 requires position 0 to be entered into the hash table (indicated with a $*$ under it) then to advance the current position by 1.

```
   ⇓
0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19
b  c  d  e  a  b  c  d  a  b  c  d  a  b  c  d  e  f  g  h
*
```

This process continues until position 8 is reached. At that time, we have this configuration:

```
                        ⇓
0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19
b  c  d  e  a  b  c  d  a  b  c  d  a  b  c  d  e  f  g  h
*  *  *  *  *  *  *  *
```

Now the rest of the sequence is "abcdabcdedfg". The longest possible match to some part previously processed is "abcdabcd" which starts at location 4. Case 2 dictates entering the last 3 positions of the match (i.e., 13, 14, 15) into the hash table, then moving the current position forward by the length of the match. Thus the current position becomes 16 in this example.

```
                                          ⇓
0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19
b  c  d  e  a  b  c  d  a  b  c  d  a  b  c  d  e  f  g  h
*  *  *  *  *  *  *  *                 *  *  *
```

The final step is to match "efgh" and that fails so the last mark is on position 16. The current position moves to position 17 which now does not have enough data left so the algorithm stops.

```
                                             ⇓
0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19
b  c  d  e  a  b  c  d  a  b  c  d  a  b  c  d  e  f  g  h
*  *  *  *  *  *  *  *                 *  *  *  *
```

Note that the above matching algorithm will actually find the longest match if indexes are kept for every location in the string. The skip in step 2b prevents the algorithm from being able to always find the longest prefix; however, this rule saves considerable processing time and memory space. In fact, it is easy to see from the above hash table construction rules that the space requirement is directly proportional to the output. The more compressible a target data set is, the faster it is to compress it.

Difference Encoding In order to minimize the output generated, the block-move list generated above must be encoded. The output of *vdelta* consists of two types of instructions: **add** and **copy**. The **add** instruction has the length of the data followed by the data itself. The **copy** instruction has the size of the data followed by its address. Two caches are maintained as references to minimize the space required to store this address information.

Each instruction is coded starting with a control byte. Eight bits of the control byte are divided into two parts. The first 4 bits represent numbers from 0 to 15, each of which defines a type of instruction and a coding of some auxiliary information. Below is an enumeration of the first 10 values of the first 4 bits:

0: an **add** instruction,
1,2,3: a **copy** instruction with position in the **QUICK** cache,
4: a **copy** instruction with position coded as an absolute offset from the beginning of the file,
5: a **copy** instruction with position coded as an offset from current location, and
6,7,8,9: a **copy** instruction with position in the **RECENT** cache.

For the **add** instruction and the **copy** instructions above, the second 4 bits of the control byte, if not zero, codes the size of the data involved. If these bits are 0, the respective size is coded as a subsequent sequence of bytes.

The above mentioned caches—**QUICK** and **RECENT**—enable more compact coding of file positions. The **QUICK** cache and is an array of size 768 ($3*256$). Each index of this array contains the value p of the position of a recent **copy** instruction such that p modulo 768 is the array index. This cache is updated after each **copy** instruction is output (during coding) or processed (during decoding). A **copy** instruction of type 1, 2, or 3 will be immediately followed by a byte whose value is from 0 to 255 that must be added to 0, 256 or 512 respectively to compute the array index where the actual position is stored. The **RECENT** cache is an array with 4 indices storing the most recent 4 copying positions. Whenever a **copy** instruction is output (during coding) or processed (during decoding), its copying position replaces the oldest position in the cache. A **copy** instruction of type 6, 7, 8, or 9 corresponds to cache index 1, 2, 3, or 4 respectively. Its copying position is guaranteed to be larger than the position stored in the corresponding cache index and only the difference is coded.

It is a result of this encoding method that an **add** instruction is never followed by another **add** instruction. Frequently, an **add** instruction has data size less than or equal to 4 and the following **copy** instruction is also small. In such cases, it is advantageous to merge the two instructions into a single control byte. The values from 10 to 15 of the first 4 bits code such merged pairs of instructions. In such a case, the first 2 bits of the second 4 bits in the control byte code the size of the **add** instruction and the remaining 2 bits code the size of the **copy** instruction. Below is an enumeration of the values from 10 to 15 of the first 4 bits:

10: a merged **add/copy** instruction with copy position coded as itself,

11: a merged **add/copy** instruction with copy position coded as difference from the current position,
12,13,14,15: a merge **add/copy** instruction with copy position coded from a **RECENT** cache.

In order to elucidate the overall encoding scheme, consider the following files:

Version1: a b c d a b c d a b c d e f g h
Version2: a b c d x y x y x y x y b c d e f

The block-move output would be

1. **copy** 4 0		01000100 0
2. **add** 2 "xy"	which encodes to	00000010 "xy"
3. **copy** 6 20	(instruction in binary)	01000110 20
4. **copy** 5 9		01000101 9

Note that the third instruction copies from Version2. The address 20 for this instruction is $16 + 4$ where 16 is the length of Version1. Note also that the data to be copied is also being reconstructed. That is, *vdelta* knows about periodic sequences.

This output encoding is independent of the way the block-move lists are calculated, thus *bdiff* could be modified to use this encoding and *vdelta* could be modified to use splay coding.

4 Method

In order to test the various differencing algorithms, a large data set was needed. Thanks to the Free Software Foundation, quite a number of software projects are available in successive versions. The authors chose two versions of GNU *emacs*—19.28 and 19.29—and two versions of GNU *gcc*—2.7.0 and 2.7.1—as their test suite. These versions provide a broad spectrum of variation between one revision of any given file and the next.

Both versions of GNU *emacs* and GNU *gcc* were compiled so that successive revisions of object files were also available. A Longest Common Subsequence was computed for each pair. Then each algorithm was run on each pair of files. Files that existed in one version and not the other and files that did not differ at all were eliminated. All this was done on a DEC Alpha system.

The algorithms chosen were UNIX *diff -n* (as used by RCS), UNIX *diff* followed by compressing the results with *gzip*, *bdiff*, and *vdelta*. The files were broken up into three types: text files (mostly C and Elisp code), byte compiled Elisp code (ELC files), and object files. Since UNIX *diff* was not designed to work with non-printable characters, UNIX *uuencode* was used to remap problematic characters. UNIX *uuencode* was chosen, since it is used in some extension to RCS to provide for binary revisioning.

Each algorithm was run with each file pair both forward and reverse, e.g. revision 19.28 then 19.29 and revision 19.29 then 19.28. This was done so that the effect of differences where one file is much smaller than the other could be averaged out. Removing large sections from one file results in a small delta and adding large sections results in a large delta. In practice, this phenomena is "averaged out" in revisioning systems since one revision must be stored in its entirety.

Two types of data were collected: the size of the delta file and the time needed to encode and decode each pair. Since no dedicated decoder is available for UNIX *diff -n* files, RCS was used to time *diff -n* encoding and decoding. (The authors tried using *diff -e* and *ed*, but that proved to be ridiculously slow.) UNIX *wc* was used to measure file sizes and UNIX time was used to measure duration. Byte count and user plus system time are used to present the results below.

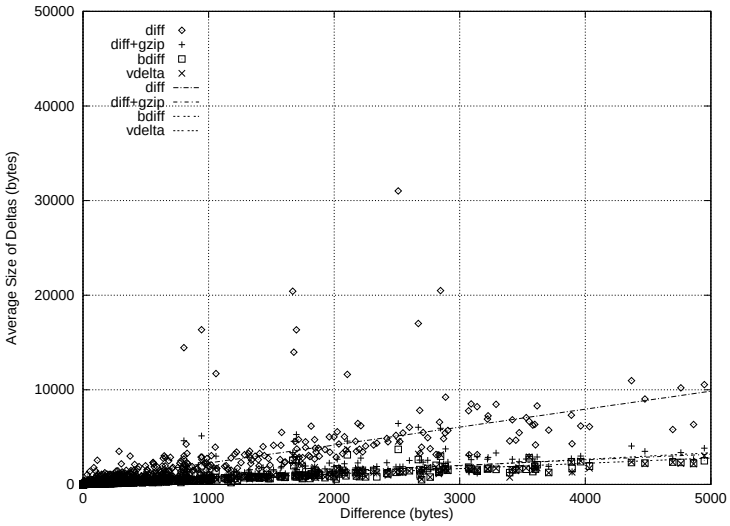
5 Results

There are two important measures for compression algorithms: the resultant compression ratio and the execution speed. The authors present the results of their study in graphic form below. These plots are presented using *gnuplot*. Results are given separately for text files, ELC files, and object files, since the behavior of some of the algorithms differ for these classes. ELC files were not combined with object files because they are mostly text.

5.1 Effectiveness

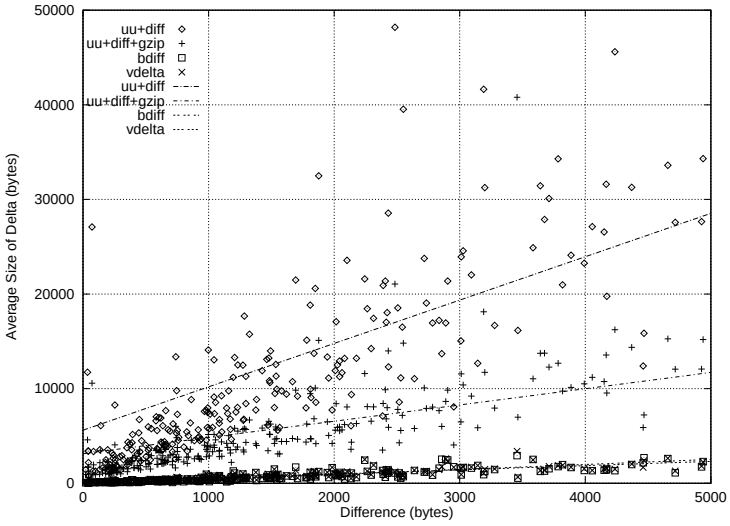
Since the *longest common subsequence* algorithm is known to produce optimal results (though at a great speed penalty), the *lcs* results are used as a point of comparison below. Specifically, the authors define the **difference** between two files as the average size of the two minus the LCS. An alternative would have been to just examine file sizes and compare them to the sum of the delta sizes. Although this number is also interesting, the LCS comparison used here sheds more light on the correlation between the actual changes between two revisions and the size of the delta. So, in each graph, a point is plotted for each file pair using each algorithm. Then a line is plotted for each algorithm that depicts the linear regression of all the data points using the standard least common squares algorithm.

In the first set of three graphs in figures 1, 2, and 3, the average size of the forward delta and reverse delta is plotted against the **difference**. The **x** axis is simply the average size of each pair minus the LCS size. The *y* axis is the average size of the forward *diff* and the reverse *diff* output. Here one can see how much better *bdiff* and *vdelta* correlate to the **difference** than *diff* and *diff* with *gzip*. Though *diff* with *gzip* performs as well as *bdiff* and *vdelta* for text files (figure 1), it performs much more poorly for ELC and object files. The outlying points correspond to files where the **difference** is much smaller than the file size. All



Correlations: diff 0.952; diff+gzip 0.976; bdiff 0.962; vdelta 0.976

Fig. 1. Plot of Delta Size for Text Files



Correlations: uu+diff 0.805; uu+diff+gzip 0.804; bdiff 0.949; vdelta 0.959

Fig. 2. Plot of Delta Size for ELC Files

algorithms performed more poorly on the object file set (figure 3) than on the other sets.

The next set of three graphs in figures 4, 5, and 6 presents the same data as a log-log plot. The logarithmic scales permit the presentation of a much greater range of **differences** and delta sizes. As expected, the linear regression lines are not straight due to their nonzero y-intercepts. This is caused by the constant factor in execution time that is most likely related to program startup time. Here one can see the low end of the graph in more detail. The separation between the data points for the different algorithms can be seen more easily and the band for *diff* with *gzip* is much closer for those of *bdiff* and *vdelta* for text files than for the other categories.

The final set of three graphs in figures 7, 8, and 9 presents the average compression ratio against the ratio of **difference** to average file size. Here the x axis is given as one minus the LCS size divided by the average file size in each pair. This expresses how much the two files differ as a ratio. The y axis is the size of the delta produced by a given algorithm divided by the average file size for the pair. This expresses the size of the delta as a ratio to the file size. Here it becomes clear how badly *uuencode* disrupts the UNIX *diff* algorithm. Good correlation is obtained for *bdiff* and *vdelta*, whereas *diff* and *diff* with *gzip* appear to be independent of the **difference** ratio.

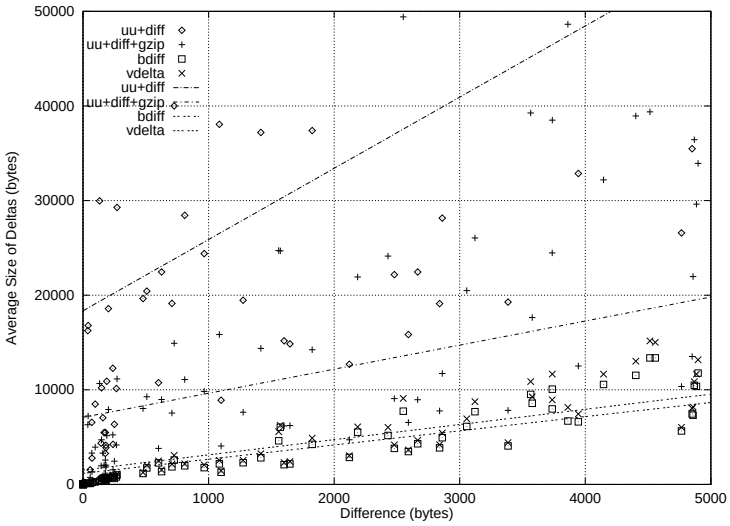
All these graphs show clear trends in the performance of *diff*, *diff* + *gzip*, *bdiff*, and *vdelta*.

5.2 Efficiency

Here the speed of both compression and decompression are given. Time is given as a sum of system and user time. This is plotted against the average file size. The first three plots in figures 10, 11, and 12 give encoding times in seconds and the remaining three plots in figures 13, 14, and 15 show decoding times in seconds. The scale is not the same for all plots, but the aspect ratio is held constant in each group. The reader should note the change in aspect ratio between the encode plots and the decode plots. Decoding is much faster for all algorithms. Though the relative performance for the others varies between encoding and decoding, *vdelta* is faster than all other algorithms for both.

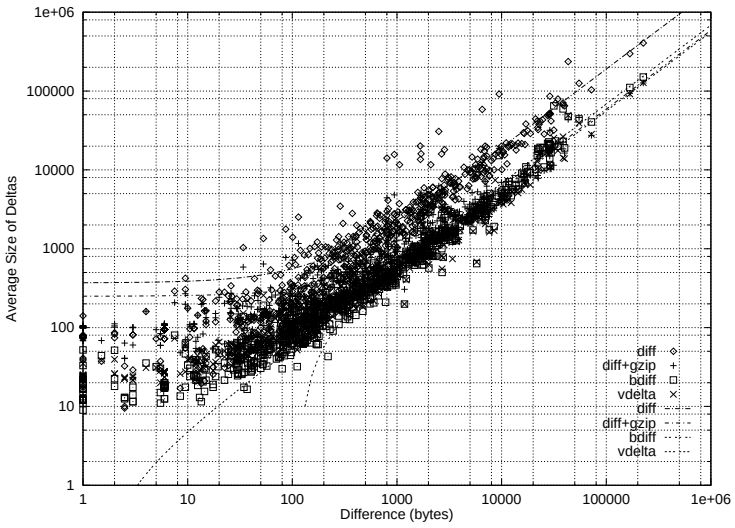
6 Conclusion

Vdelta is the best algorithm overall. Its coding and decoding performance is high enough to be used for interactive applications. For example, it could be used to improve performance of raster display updates over a relatively slow networks links. Though *bdiff* generates output that is comparable in size to *vdelta*, *vdelta* is much faster. Both *vdelta* and *bdiff* result in delta sizes that correlate well with the **difference**. This is not true for *diff*. In the best case—text files—*diff* only reaches the effectiveness of *vdelta* and *bdiff* when it is combined with *gzip*. Using *uuencode* is not a good idea for binary files, since it breaks *diff*'s algorithm for



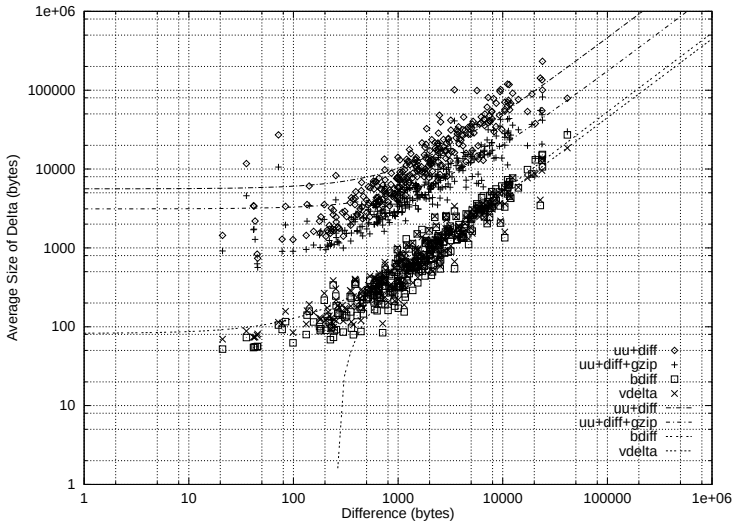
Correlations: uu+diff 0.847; uu+diff+gzip 0.857; bdiff 0.959; vdelta 0.937

Fig. 3. Plot of Delta Size for Object Files



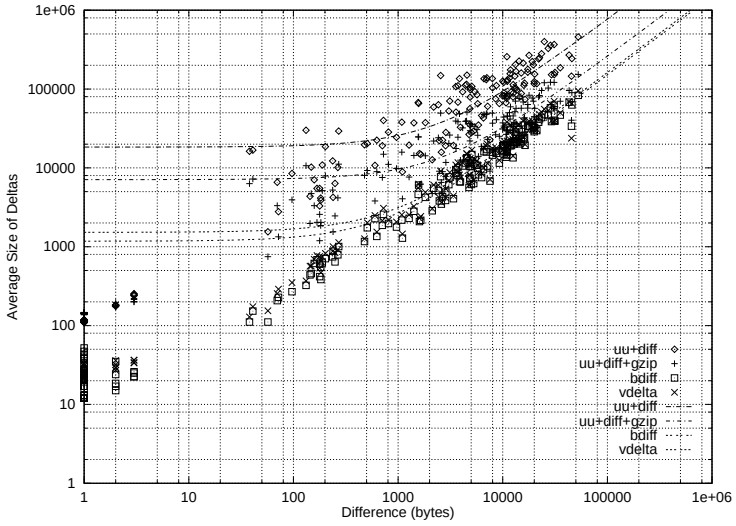
Correlations: diff 0.952; diff+gzip 0.976; bdiff 0.962; vdelta 0.976

Fig. 4. Log Plot of Delta Size for Text Files



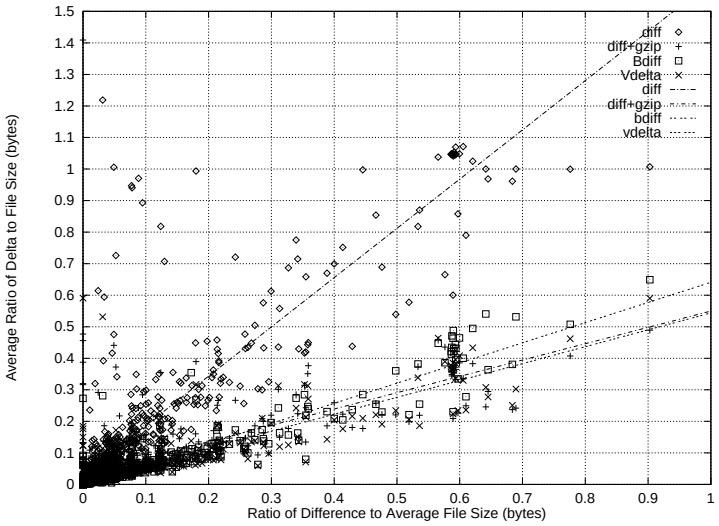
Correlations: uu+diff 0.805; uu+diff+gzip 0.804; bdiff 0.949; vdelta 0.959

Fig. 5. Log Plot of Delta Size for ELC Files



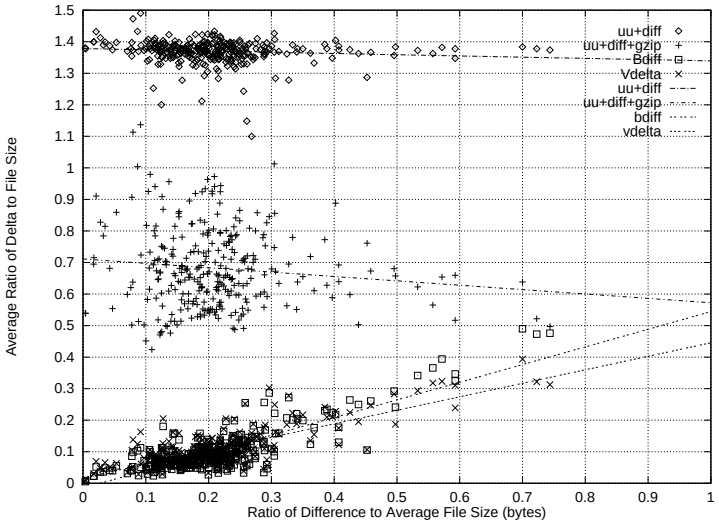
Correlations: uu+diff 0.847; uu+diff+gzip 0.857; bdiff 0.959; vdelta 0.937

Fig. 6. Log Plot of Delta Size for Object Files



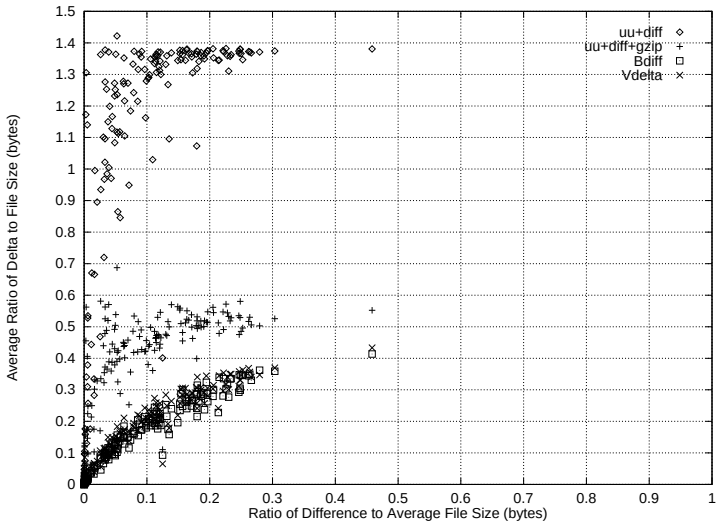
Correlations: diff 0.877; diff+gzip 0.545; bdiff 0.947; vdelta 0.870

Fig. 7. Plot of Compression Ratio for Text Files



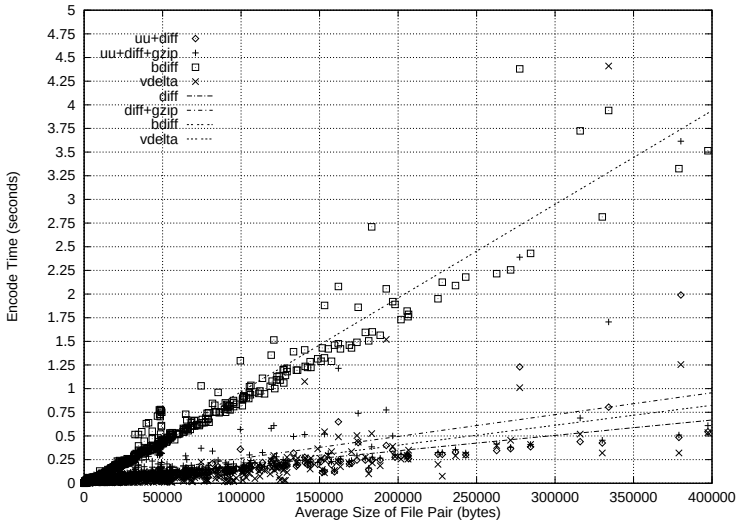
Correlations: uu+diff -0.111; uu+diff+gzip -0.118; bdiff 0.832; vdelta 0.778

Fig. 8. Plot of Compression Ratio for ELC Files



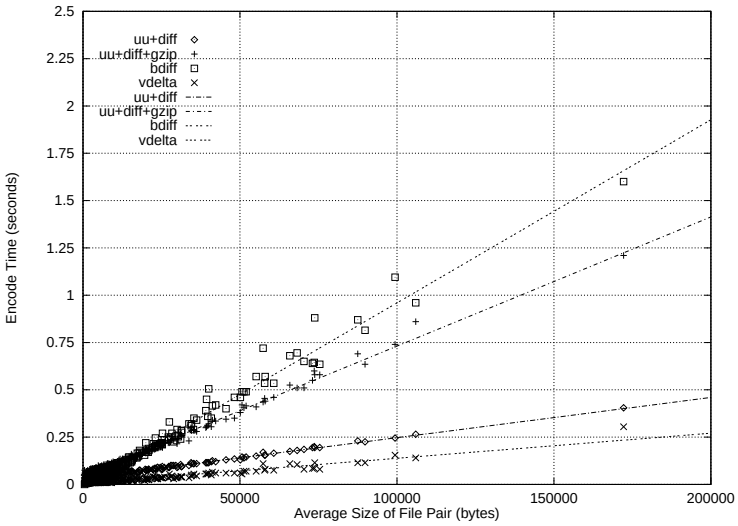
Correlations: uu+diff 0.742; uu+diff+gzip 0.736; bdiff 0.958; vdelta 0.945

Fig. 9. Plot of Compression Ratio for Object Files



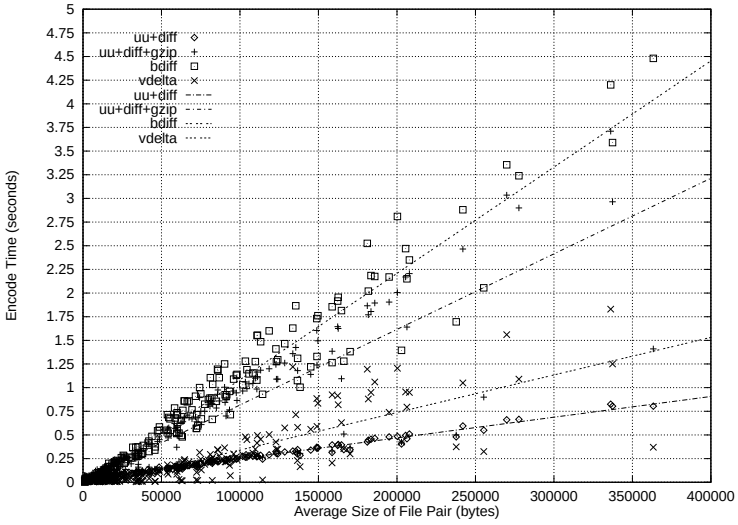
Correlations: diff 0.813; diff+gzip 0.672; bdiff 0.971; vdelta 0.599

Fig. 10. Plot of Encoding Time for Text Files



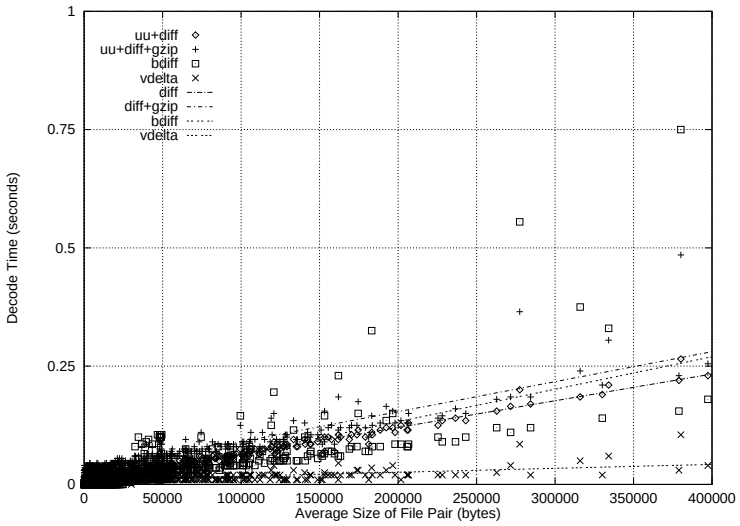
Correlations: uu+diff 0.993; uu+diff+gzip 0.996; bdiff 0.992; vdelta 0.965

Fig. 11. Plot of Encoding Time for ELC Files



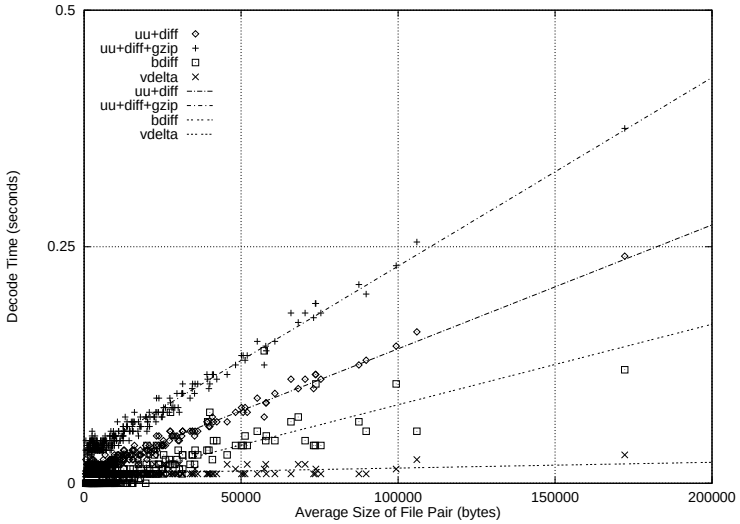
Correlations: uu+diff 0.994; uu+diff+gzip 0.875; bdiff 0.973; vdelta 0.855

Fig. 12. Plot of Encoding Time for Object Files



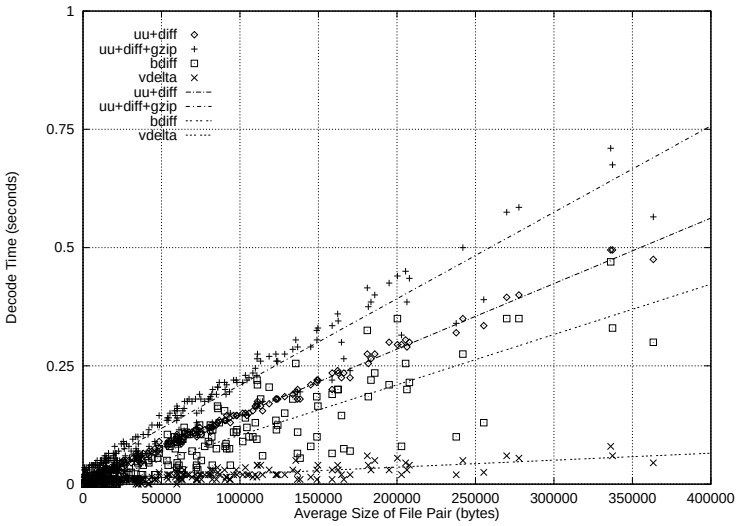
Correlations: diff 0.987; diff+gzip 0.927; bdiff 0.775; vdelta 0.612

Fig. 13. Plot of Decoding Time for Text Files



Correlations: uu+diff 0.985; uu+diff+gzip 0.987; bdiff 0.868; vdelta 0.395

Fig. 14. Plot of Decoding Time for ELC Files



Correlations: uu+diff 0.997; uu+diff+gzip 0.976; bdiff 0.883; vdelta 0.787

Fig. 15. Plot of Decoding Time for Object Files

detecting unchanged sequences. This property was expected, because *uuencode* essentially removes all natural newlines and adds new ones at constant intervals. This means that only changes that do not modify the positions of unmodified characters or change the file length by an exact multiple of the constant interval can be effectively processed by *diff*. *Diff*'s only advantage is that it provides human readable differences; however, this can be added to the others as well.

7 Future Work

The result of this paper apply only to one dimensional data. Other studies should be done for two dimensional data such as for pictures. In addition, this test suite could be used to fine tune both *bdiff* and *vdelta*, to determine what effect block-move list encoding has on run time and delta size.

References

1. James A. Gosling. A redisplay algorithm. In *Proc. of the ACM SIGPLAN/SIGOA Symposium on Text Manipulation*, pages 123–129, 1981.
2. James W. Hunt and M.D. McIllroy. An algorithm for differential file comparison. Technical Report Computing Science Technical Report 41, Bell Laboratories, June 1976.

3. James W. Hunt and Thomas G. Szymanski. A fast algorithm for computing longest common subsequences. *Communications of the ACM*, 20(5):350–353, May 1977.
4. Douglas W. Jones. Application of splay trees to data compression. *Communications of the ACM*, 31(8):996–1007, August 1988.
5. David G. Korn and Kiem-Phong Vo. Vdelta: Efficient data differencing and compression. *In preparation*, 1995.
6. E. M. McCreight. A space economical suffix tree construction algorithm. *Journal of the ACM*, 32:262–272, 1976.
7. Webb Miller and Eugene W. Meyers. A file comparison program. *Software—Practice and Experience*, 15(11):1025–1039, November 1985.
8. Narao Nakatsu, Yahiko Kambayashi, and Shuzo Yajima. A longest common subsequence algorithm for similar text strings. *Acta Informatica*, 18:171–179, 1982.
9. Wolfgang Obst. Delta technique and string-to-string correction. In *Proc. of the First European Software Engineering Conference*, pages 69–73. AFCET, Springer Verlag, September 1987.
10. Marc J. Rochkind. The source code control system. *IEEE Transactions on Software Engineering*, SE-1(4):364–370, December 1975.
11. Walter F. Tichy. The string-to-string correction problem with block moves. *ACM Transactions on Computer Systems*, 2(4):309–321, November 1984.
12. Walter F. Tichy. RCS — a system for version control. *Software—Practice and Experience*, 15(7):637–654, July 1985.
13. Kiem-Phong Vo. A prefix matching algorithm suitable for data compression. *In preparation*, 1995.
14. J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Trans. on Information Theory*, IT-24(5):5306, September 1978.

This article was processed using the LaTeX macro package with LLNCS style